Information Hiding in Software with Mixed Boolean-Arithmetic Transforms

Yongxin Zhou, Alec Main, Yuan X. Gu, and Harold Johnson

Cloakware Inc., USA
{yongxin.zhou,alec.main,yuan.gu,harold.johnson}@cloakware.com

Abstract. As increasingly powerful software analysis and attack tools arise, we need increasingly potent software protections. We generate an *unlimited* supply of obscuring transforms via mixed-mode computation over Boolean-arithmetic (MBA) algebras corresponding to real-world functions and data. Such transforms resist reverse engineering with existing advanced tools and create NP-hard problems for the attacker. We discuss broad uses and concrete applications to AACS key hiding and software watermarking.

1 Introduction

With the increasing power of software analysis and attack tools and the ubiquity of open operating systems, ever stronger software data- and algorithm-hiding mechanisms are essential. (For existing protections, see, e.g., [3,4,5,6,7,10,18,21].)

We introduce Boolean-arithmetic (BA) algebras which model real-world software computation. Modern ALUs use (1) 2's complement arithmetic on n-bit words, which maps to the modular ring $\mathsf{Z}/(2^n)$, and (2) bitwise operations over B^n where $\mathsf{B} = \{0, 1\}$. Combining (1) and (2) gives the BA-algebra $\mathsf{BA}[n]$, over which we define highly simplification-resistant mixed Boolean-arithmetic (MBA) obfuscating transforms with NP-hard fragment recognition, for which we provide unlimited-volume generators, ensuring an ever-growing MBA protections database.

MBA transforms based on MBA expressions, MBA identities and invertible functions can be used to transform software code: functionality is preserved, but secret constants, intermediate values, and algorithms are hidden from static and dynamic reverse engineering and analysis. Perimeter software protection methods, that ultimately allowed the data or algorithm to appear in memory cannot provide a similar level of protection.

Moreover, transformed software simultaneously occupies multiple mathematical domains, creating worst-case NP-hard problems for the attacker, and making them highly resistant even to advanced analytical tools such as **Maple**$^{\text{TM}}$ or **Mathematica**$^{\text{TM}}$. The unlimited supply of MBA transforms can be generated ensuring an ever-growing search space to protect against new tools for analyzing and attacking transformed software.

S. Kim, M. Yung, and H.-W. Lee (Eds.): WISA 2007, LNCS 4867, pp. 61–75, 2007.

We provide practical threat scenarios. We then introduce MBA transforms and their use for protection against such threats. Lastly, we provide practical examples of transformed algorithms.

2 Motivating Scenarios

2.1 Naïve Code

In Fig. 1(A)'s scenario, both algorithm and constant data C are unprotected, permitting algorithmic reverse engineering by static analysis and extraction of C by searching the executable file.

This is the normal state of software as written, compiled, and delivered.

2.2 Hiding Constants from Static Analysis

Hiding C in code gives us Fig. 1(B). The algorithm is still exposed to static analysis, but C (created at runtime) no longer appears in the executable file.

This (without MBA methods) was the situation in the Advanced Access Control System (AACS: see Fig. 2), a copy protection system adopted for HD-DVD™ and **BluRay**™ optical discs, which was hacked[22] by various parties by initially obtaining the title key, K_t, from memory, then working back through the chain to obtain earlier keys from memory, such as the media key, K_m. Since the algorithms are public, obtaining the keys allows for off-line decryption of the licensed content by an application that does not comply with licensing terms (e.g. writes decrypted content to a file). Obtaining the higher level keys allows the others to be calculated, making the memory-key-extraction step simpler. As in Fig. 1(B), while the software manufacturers made an effort to hide the device keys, they did not prevent intermediate values from appearing in memory.

2.3 Hiding Constants and Algorithms from Dynamic Analysis

Finally, in Fig. 1(C)'s scenario, we protect the value of C and its associated code by applying code and data transforms[3,10,21]. The constant code generates C^T (C under transformation T) with corresponding changes to the algorithm. Even if a dynamic attack recovers C^T, an attacker must reverse-engineer of the transformed algorithm to find T. E.g. if C is a cryptographic key, offline decryption is infeasible until T is found or the algorithm code is lifted, but transform diversity[21] makes lifting less useful. Moreover, lifting could be made difficult if the code surrounding decryptions were mathematically intermixed with the decryptions using MBA transforms.

Example: AACS. The above is the approach we would recommend for systems such as AACS: using MBA transforms would ensure that only *transformed* key values appear in memory or registers during computation, with a transform not useful for offline decryption without significant additional analysis.

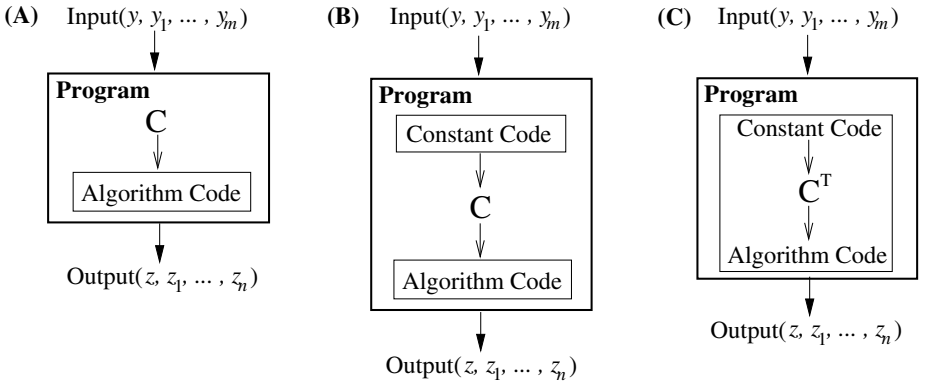

Fig. 1. Basic Scenarios

A possible attack point would be to compare known title keys (obtained from previously hacked titles) with the corresponding transformed keys. To render such an attack ineffective, the transformation must be sufficiently complex that determining its functionality from such isolated plain-key-to-transformed-key instances is infeasibly hard for the system's anticipated attackers.

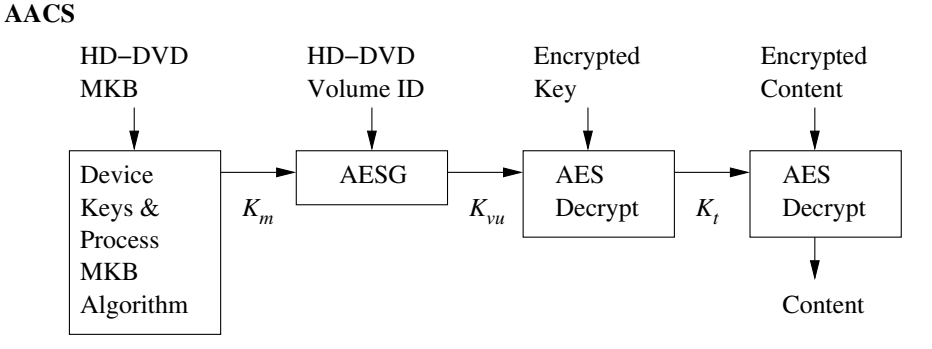

Fig. 2. AACS

Example: Software Watermarking. Another possible use for key and algorithm hiding is software watermarking in order to prove ownership of the software or intellectual property contained therein if the software is used without a license. Software watermarking is not to be confused with steganographic watermarking, where a message is hidden images or an audio or video stream, although some systems (e.g., DRM systems) permit both forms of protection to be used (software watermarking for DRM components, steganographic watermarking for the content they manage).

```
Q: if I = K then
       K = E(K)
       W = emit_watermark (P, K)
       output W
   else
       P(I)
```

Fig. 3. Watermarking

In §4.2, we will provide a simple example where software function is dependent on a secret key K which can be used to prove software ownership. MBA transforms effectively protect the software watermark algorithm by making it very hard to discover K and by interlocking (see §3.4) the constant code producing K and the watermark algorithm (see Fig. 3).

3 Mixed Boolean-Arithmetic (MBA) Transforms

3.1 Basic Definitions

Microprocessor arithmetic logic units (ALUs) use arithmetic operations including addition $+$, subtraction $-$ (whence comparisons $<, \leq, =, \geq, >$), multiply $\cdot$, left shift $\ll$, arithmetic right shift $\gg^s$, and logical right shift $\gg$. Bitwise operations include *exclusive-or* $\oplus$, *inclusive-or* $\vee$, *and* $\wedge$, and *not* $\neg$. With n-bit 2's complement integers, arithmetic operations are in integer modular ring $\mathsf{Z}/(2^n)$. Bitwise operations operate over Boolean algebra $(\mathsf{B}^n, \wedge, \vee, \neg)$. All above computations are captured in the BA-algebra, $\mathsf{BA}[n]$.

Definition 1. *With n a positive integer and $\mathsf{B} = \{0, 1\}$, the algebraic system $(\mathsf{B}^n, \wedge, \vee, \oplus, \neg, \leq, \geq, >, <, \leq^s, \geq^s, >^s, <^s, \neq, =, \gg^s, \gg, \ll, +, -, \cdot)$, where $\ll, \gg$ denote left and right shifts, $\cdot$ (or juxtaposition) denotes multiply, and signed compares and arithmetic right shift are indicated by ^{s}, is a* Boolean-arithmetic algebra (BA-algebra)*, $\mathsf{BA}[n]$. n is the* dimension *of the algebra.*

$\mathsf{BA}[n]$ includes the Boolean algebra $(\mathsf{B}^n, \wedge, \vee, \neg)$, the integer modular ring $\mathsf{Z}/(2^n)$, Galois field $\mathrm{GF}(2^n)$, and *p-adic* numbers[19]. The first two structures are most used in real applications and form the basis of our techniques for hiding keys (constants) and operations over $\mathsf{BA}[n]$.

Definition 2. *With $\mathsf{BA}[n]$ a BA-algebra and t a positive integer, a function f: $(\mathsf{B}^n)^t \mapsto \mathsf{B}^n$ of the form*

$$\sum_{i \in I} a_i \left(\prod_{j \in J_i} e_{i,j}(x_1, \ldots, x_t) \right),$$

where a_i are constants, $e_{i,j}$ are bitwise expressions of variables $x_1, \ldots, x_t$ over B^n, and $I, J_i \subset Z$, are finite index sets, $\forall i \in I$, is a polynomial mixed Boolean-arithmetic expression*, abbreviated to* a polynomial MBA expression*. (If each of $x_1, \ldots, x_t$ is itself a polynomial* MBA *expression of other variables, the composed function is likewise a polynomial* MBA *expression over B^n.) Each non-zero summand in the expression is a* term*. A* linear MBA expression *is a polynomial* MBA *expression of the form*

$$\sum_{i \in I} a_i \, e_i(x_1, \ldots, x_t),$$

where e_i are bitwise expressions of $x_1, \ldots, x_t$ and a_i are constants.

Two examples of polynomial MBA expressions over $\mathsf{BA}[n]$ are:

$$\begin{aligned} f(x,y,z,t) &= 8458(x \vee y \wedge z)^3 \, ((xy) \wedge x \vee t) + x + 9(x \vee y)yz^3, \\ f(x,y) &= x + y - (x \oplus (\neg y))) - 2(x \vee y) + 12564. \end{aligned}$$

The latter is a linear MBA expression. As indicated in [20], all integer comparison operations can be represented by polynomial MBA expressions with results in their most significant bit (MSB). For example, the MSB of

$$(x - y) \oplus ((x \oplus y) \wedge ((x - y) \oplus x))$$

is 1 if and only if $x <^s y$.

3.2 Linear MBA Identities and Expressions

We now show the existence of an unlimited number of linear MBA identities. We use truth tables, where the relationship of variables of the expression in the table and conjuncts is shown by example in Table 1.

Theorem 1. *Let n, s, t be positive integers, let x_i be variables over B^n for $i = 1, \ldots, t$, let e_j be bitwise expressions on x_i's for $j = 0, \ldots, s-1$. Let $e = \sum_{j=0}^{s-1} a_j \, e_j$ be a linear* MBA *expression, where a_j are integers, $j = 0, \ldots, s-1$. Let f_j be the deduced Boolean expression from e_j, and let $(v_{0,j}, \ldots, v_{i,j}, \ldots, v_{2^t-1,j})^{\mathsf{T}}$ be the column vector of the truth table of f_j, $j = 0, \ldots s-1$, and $i = 0, \ldots, 2^t - 1$. Let $A = (v_{i,j})_{2^t \times s}$, be the {0,1}-matrix of truth tables over $Z/(2^n)$. Then $e = 0$ if and only if the linear system $AY = 0$ has a solution over ring $Z/(2^n)$, where $Y_{s \times 1} = (y_0, \cdots, y_{s-1})^{\mathsf{T}}$ is a vector of s variables over $Z/(2^n)$.*

Proof. If $e = 0$, $(a_0, a_1, \cdots, a_{s-1})^{\mathsf{T}}$ is plainly a solution of the linear system.

Assume a solution exists. Let z_{ji} represent the $i-$th bit value of e_j, $j = 0, 1, \cdots, s-1$, $i = 0, 1, \cdots, n-1$. Truth tables run over all inputs, so row vectors of matrix A run over all values of i-th bit vector $(z_{0,i}, \ldots, z_{s-1,i})$ in the Boolean expressions, and via the above solution, $\sum_{j=0}^{s-1} a_j z_{j,i} = 0$, for $i = 0, \ldots, n-1$.

From the arithmetic point of view, $e_j = \sum_{i=0}^{n-1} z_{j,i} 2^i$. Thus we have

$$\sum_{j=0}^{s-1} a_j e_j = \sum_{j=0}^{s-1} \sum_{i=0}^{n-1} a_j z_{j,i} 2^i = \sum_{i=0}^{n-1} \sum_{j=0}^{s-1} a_j z_{j,i} 2^i = \sum_{i=0}^{n-1} 2^i (\sum_{j=0}^{s-1} a_j z_{j,i}) = 0,$$

as required. □

By this theorem, any $\{0,1\}$-matrix $(v_{ij})_{2^t \times s}$ with linearly dependent column vectors generates a linear MBA identity of t variables over B^n. For example, the $\{0,1\}$-matrix

$$A = \begin{pmatrix} 0\,0\,0\,1\,1 \\ 0\,1\,1\,1\,1 \\ 1\,0\,1\,1\,1 \\ 1\,1\,1\,0\,1 \end{pmatrix}$$

with column-vector truth-tables for $f_0(x,y) = x, f_1(x,y) = y, f_2(x_y) = x \vee y$, $f_3(x,y) = \neg(x \wedge y), f_4(x,y) = 1$, respectively, with solution $(1,1,-1,1,-1)^\mathsf{T}$ over $\mathsf{Z}/(2^{32})$, generates linear MBA identity $x + y - (x \vee y) + (\neg(x \wedge y)) - (-1) = 0$, using $x_1 = x$, $x_2 = y$, and $-1 =$ all 1's. The interested reader can write **C** code to verify the identity for any $x, y \in \mathsf{B}^{32}$ by defining x and y as unsigned or signed 32-bit integers. Other linear MBA identities can be found in [20].

The following theorem states that any Boolean function has non-trivial linear MBA expressions. It allows us to embed hard Boolean functions into MBA transforms.

Table 1. Truth Table for $x_1 \vee (x_2 \oplus (\neg x_3))$

Conjunction	Binary	Result
$(\neg x_1) \wedge (\neg x_2) \wedge (\neg x_3)$	000	1
$(\neg x_1) \wedge (\neg x_2) \wedge (\ \ x_3)$	001	0
$(\neg x_1) \wedge (\ \ x_2) \wedge (\neg x_3)$	010	0
$(\neg x_1) \wedge (\ \ x_2) \wedge (\ \ x_3)$	011	1
$(\ \ x_1) \wedge (\neg x_2) \wedge (\neg x_3)$	100	1
$(\ \ x_1) \wedge (\neg x_2) \wedge (\ \ x_3)$	101	1
$(\ \ x_1) \wedge (\ \ x_2) \wedge (\neg x_3)$	110	1
$(\ \ x_1) \wedge (\ \ x_2) \wedge (\ \ x_3)$	111	1

Theorem 2. *Let e be a bitwise expression of m variables over B^n. Then e has a non-trivial linear* MBA *expression. That is $e = \sum_{i=0}^{2^m-1} a_i e_i$, where each $a_i \in \mathsf{B}^n$, each e_i is a bitwise expression of the m variables, and $e \neq e_i$ in B^n, for $i = 0, 1, \ldots, 2^m - 1$.*

Proof. Let $P_{2^m \times 1}$ be the column vector of the truth table of the deduced Boolean expression from e. Pick an invertible $\{0,1\}$-matrix $A_{2^m \times 2^m}$ over $\mathsf{Z}/(2^n)$. If a column vector of A is P, add another column vector to it. Then we have an invertible matrix A with all column vectors distinct from P. Suppose the solution of linear equation $AY = P_{2^m \times 1}$ is $Y = (y_0, y_1, \ldots, y_{2^m-1})^\mathsf{T}$. Let matrix $Q_{2^m \times (2^m+1)} = (P, A)$, and $X_{(2^m+1) \times 1} = (-1, y_0, y_1, \ldots, y_{2^m-1})^\mathsf{T}$. Because $AY = P$, it is easy to show that $QX = 0$.

Following the disjunctive normal form (or any standard form) of Boolean functions, for each column vector A_i of matrix A we have a unique Boolean expression e_i of m variables with A_i being its truth table, $i = 0, 1, \ldots, 2^m - 1$.

Since $QX = 0$, Theorem 1 gives a linear MBA identity $e - \sum_{i=0}^{2^m-1} y_i\, e_i = 0$. By our choice of A, $e_i \neq e$ for all i. □

3.3 Permutation Polynomials and Other Invertible Functions

We use polynomial functions over $\mathsf{Z}/(2^n)$ to generate polynomial MBA expressions. We need invertible polynomials[15,13,12], and both the polynomial and its inverse must be of limited degree so that polynomial code transformations are efficient. We now show that there are many such invertible polynomials.

Theorem 3. *Let m be a positive integer and let $P_m(Z/(2^n))$ be a set of polynomials over $Z/(2^n)$:*

$$P_m(Z/(2^n)) = \left\{ \sum_{i=0}^{m} a_i x^i \;\middle|\; \forall a_i \in Z/(2^n), a_1 \wedge 1 = 1, a_i^2 = 0, i = 2, \ldots, m \right\}.$$

Then $(P_m(Z/(2^n)), \circ)$ is a permutation group under the functional composition operator $\circ$. For every element $f(x) = \sum_{i=0}^{m} a_i x^i$, its inverse $g(x) = \sum_{j=0}^{m} b_j x^j$ can be computed by

$$\begin{aligned}
b_m &= -a_1^{-m-1} a_m, \\
b_k &= -a_1^{-k-1} a_k - a_1^{-1} \textstyle\sum_{j=k+1}^{m} \binom{j}{k} a_0^{j-k} A_j, m-1 \geq k \geq 2, \\
b_1 &= a_1^{-1} - a_1^{-1} \textstyle\sum_{j=2}^{m} j a_0^{j-1} A_j, \\
b_0 &= -\textstyle\sum_{j=1}^{m} a_0^j b_j,
\end{aligned}$$

where $A_m = -a_1^{-m} a_m$, and A_k are recursively defined by

$$A_k = -a_1^{-k} a_k \; - \sum_{j=k+1}^{m} \binom{j}{k} a_0^{j-k} A_j, \quad \textit{for } 2 \leq k < m.$$

Proof. By definition of $\mathsf{P}_m(\mathsf{Z}/(2^n))$, the coefficient of x is odd, and coefficients of higher degrees are even. By [15], they are permutation polynomials. To show $(\mathsf{P}_m(\mathsf{Z}/(2^n)), \circ)$ is a group, we compute all coefficients of the composition $g(x) \circ f(x) = g(f(x)) = \sum_{j=0}^{m} b_j f(x)^j$ of any two elements $f(x) = \sum_{i=0}^{m} a_i x^i$ and $g(x) = \sum_{j=0}^{m} b_j x^j \in P_m(\mathsf{Z}/(2^n))$. For any $j \in \{2, \ldots, m\}$, we have

$$\begin{aligned}
b_j f(x)^j &= \textstyle\sum_{(i_1 \cdots i_j), i_k \in \{0,\ldots,m\}, k=0,\ldots,j} b_j a_{i_1} \cdots a_{i_j} x^{i_1+\cdots+i_j} \\
&= \textstyle\sum_{(i_1 \cdots i_j), i_k \in \{0,1\}, k=0,\ldots,j} b_j a_{i_1} \cdots a_{i_j} x^{i_1+\cdots+i_j} \text{ (since } b_j a_i = 0, \forall i \geq 2) \\
&= b_j \textstyle\sum_{k=0}^{j} \binom{j}{k} a_1^k a_0^{j-k} x^k; \\
g(f(x)) &= b_0 + \textstyle\sum_{i=0}^{m} b_1 a_i x^i + \sum_{j=2}^{m} b_j \sum_{k=0}^{j} \binom{j}{k} a_1^k a_0^{j-k} x^k \\
&= b_0 + \textstyle\sum_{k=0}^{m} (b_1 a_k + \sum_{j=2, j \geq k}^{m} b_j \binom{j}{k} a_1^k a_0^{j-k}) x^k \\
&= \left(\textstyle\sum_{j=0}^{m} a_0^j b_j \right) + \left(\sum_{j=1}^{m} j a_0^{j-1} a_1 b_j \right) x \\
&+ \textstyle\sum_{k=2}^{m} \left(a_k b_1 + \sum_{j=k}^{m} \binom{j}{k} a_0^{j-k} a_1^k b_j \right) x^k.
\end{aligned}$$

Let $\sum_{i=0}^{m} c_i x^i$ denote $g(f(x))$. Then c_1 is odd since both a_1 and b_1 are odd, and all b_j, $j \geq 2$, are even. For all a_k and b_j, $k, j \geq 2, a_k^2 = b_j^2 = 0$, so we have $c_i^2 = 0, i \geq 2$. Therefore $g(f(x)) \in \mathsf{P}_m(\mathsf{Z}/(2^n))$ and $\mathsf{P}_m(\mathsf{Z}/(2^n)$ is a group.

Let us compute the inverse of $f(x)$. Assume $g(f(x)) = x$, which implies $c_1 = 1$ and $c_i = 0, i = 0, 2, \cdots, m$. From $c_m = 0$ we have $b_m = -a_1^{-m} a_m b_1$. Similarly, for all k, $m > k \geq 2$, $c_k = 0$ implies $b_k = -a_1^{-k} a_k - \sum_{j=k+1}^{m} b_j \binom{j}{k} a_0^{j-k}$. Observing that the second term of b_k is defined by $b_j, j > k$, and b_1 is a common factor of all these b_k starting at b_m, we can define $b_k = b_1 A_k$ such that all A_k can be computed from $A_m, A_{m-1}, \ldots, A_{k+1}$; i.e., $A_m = -a_1^{-m} a_m$, and $A_k = -a_1^{-k} a_k - \sum_{j=k+1}^{m} \binom{j}{k} a_0^{j-k} A_j$, from known coefficients of $f(x)$. From $c_1 = 1$ we obtain $b_1 = a_1^{-1}(1 + \sum_{j=2}^{m} j a_0^{j-1} A_j)^{-1} = a_1^{-1}(1 - \sum_{j=2}^{m} j a_0^{j-1} A_j)$. The latter identity holds because $A_k^2 = 0$ for all k, $m > k \geq 2$. Based on this formula for b_1 and the identities $b_k = b_1 A_k$, and $c_0 = 0$, we can compute all other coefficients recursively: $b_m = -a_1^{-m-1} a_m$, and for all k with $m > k \geq 2$,

$$\begin{aligned} b_k &= a_1^{-1}(1 - \textstyle\sum_{j=2}^{m} j a_0^{j-1} A_j)(-a_1^{-k} a_k - \sum_{j=k+1}^{m} \binom{j}{k} a_0^{j-k} A_j) \\ &= -a_1^{-k-1} a_k - a_1^{-1} \textstyle\sum_{j=k+1}^{m} \binom{j}{k} a_0^{j-k} A_j. \end{aligned}$$

We then easily derive $b_0 = -\sum_{j=1}^{m} a_0^j b_j$. □

$\mathsf{P}_1(\mathsf{Z}/(2^n))$ is used for data transforms in [10]. Formula of polynomial inverses of polynomials in $\mathsf{P}_2(\mathsf{Z}/(2^n))$ is given in [21]. Following Theorem 3, for cubic polynomial

$$f(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

in $\mathsf{P}_3(\mathsf{Z}/(2^n))$, its inverse is

$$\begin{aligned} f^{-1}(x) = {} & (-a_1^{-4} a_3) x^3 + (-a_2 a_1^{-3} + 3 a_0 a_1^{-4} a_3) x^2 \\ & + (a_1^{-1} + 2 a_0 a_1^{-3} a_2 - 3 a_0^2 a_1^{-4} a_3) x - a_0 a_1^{-1} - a_0^2 a_1^{-3} a_2 + a_0^3 a_1^{-4} a_3. \end{aligned}$$

The above theorem implies that all encoding, decoding and composition functions are in conveniently similar formats for generating code transforms.

Over $\mathsf{BA}[n]$, there are other types of invertible functions that can be used together with MBA identities and permutation polynomials. For example, $\mathsf{Z}/(2^n)$ invertible matrices can be composed with invertible polynomials to form polynomial matrices and matrix polynomials. The T-functions of [11,12] provide another example.

3.4 Code Transforms Via Zero and Invertible MBA Functions

Operating on transformed data requires that the data and code (see Fig. 1(A)) use corresponding transforms (see Fig. 1(C)). MBA transforms constructed using zero functions (functions returning zero irrespective of their inputs) and invertible functions achieve this while interlocking and hiding the original operations. Linear MBA identities and permutation polynomials over $\mathsf{BA}[n]$ are our main resources.

There are two basic methods to hide operation in polynomial MBA expressions. The first one is to treat code (possibly a single operation or variable) as a subexpression in a polynomial MBA zero function and replace it with the negated sum of the remaining terms (i.e., it performs an identity substitution $t_k = -\sum_{i=1}^{k-1} t_i + \sum_{i=k+1}^{n} t_i$ derived from a zero function of the form $\sum_{i=1}^{n} t_i = 0$).

The other method uses functional compositions of MBA functions and the first method. For example, given two invertible functions S and T, function f is $S^{-1} \circ S \circ f \circ T \circ T^{-1}$. The associative law of functional composition allows us to write f as $S^{-1} \circ (S \circ f \circ T) \circ T^{-1}$. Applying the first method to $(S \circ f \circ T)$ before compositions we obtain a new expression of f; i.e., we compute on 'encrypted' data with 'encrypted' functions[16,17].

A key technique is *interlocking:* (1) *reverse partially evaluate: at Y, replace a function* $f\colon A \mapsto C$ *with a function* $g\colon A \times B \mapsto C$ *such that* $g(\cdot, b) = f(\cdot)$ *but* $g(\cdot, x)$ *where* $x \neq b$ *gives a nonsense result and b (absent tampering) comprises value(s) computed at X, and then* (2) *make code at site Y highly dependent on code at site X by using values produced or obtained at X as coefficients in Y's transforms and finally* (3) *apply the remaining protections in this paper to make the interlock extremely hard to eliminate.* Removing such interlocks requires that the attacker fully understand what has been done (highly nontrivial, as we argue in §5), and tampering at X produces *chaotic* behavioral changes at Y. Thus dense interlocking renders code aggressively fragile under tampering. Interlocks also foil code-lifting.

Compositions of zero and/or invertible functions with original operations in specific orders give us desired polynomial MBA expressions, as shown in the proof below.

Proposition 1. *Let m be a positive integer. Then every operation in* BA*-algebra* $\mathsf{BA}[n] = (\mathsf{B}^n, \wedge, \vee, \oplus, \neg, \leq, \geq, >, <, \leq^s, \geq^s, >^s, <^s, \neq, =, \gg^s, \gg, \ll, +, -, \cdot)$ *can be represented by a high degree polynomial* MBA *expressions of multiple terms of m variables over* B^n.

Proof. It suffices to show that each operation can be represented by a linear MBA identity of at least two terms, since then we can apply operation and composition transforms with invertible polynomials according to Theorem 3 to obtain high degree polynomial MBA expressions of any desired number of variables.

Any single variable x over B^n can be represented by a linear MBA expression using any number of variables: this follows immediately from Theorem 2 by using x's truth table. If we have $x = \sum_{i \in I} e_i$, where e_i are bitwise expressions and $y = \sum_{s \in S} t_s$ where t_s are bitwise expressions, then $x \pm y$ is a linear MBA expression and by $\mathsf{Z}/(2^n)$ distributivity, xy is in a polynomial MBA expression. So is $\ll$: it is a multiplication. For right shifts we have $x \gg^s y = -((-x-1) \gg^s y) - 1$ and $x \gg y = -1 - ((-x-1) \gg y) - (((-1) \gg y) \oplus (-1))$. Theorem 2 indicates that all bitwise operations can be a linear MBA expression with at least two terms. Formulas in [20] express arbitrary signed and unsigned comparisons as bitwise operations with subtraction. We then apply Theorem 2 to obtain expressions of more that two terms. (This construction is one of many due to $\mathsf{BA}[n]$'s rich mathematical structure.) □

4 Protection Methods

4.1 Simple Constant Hiding Using MBA Transforms

In many applications, important information is represented by constants (keys). Here, we provide methods to turn them into executable code which computes the keys irrespective of its inputs, letting us seamlessly embed keys in applications. WLOG, we consider n-bit key hiding in $\mathsf{BA}[n]$ only: if the key size bigger than n, we simply generate multiple n-bit keys which are concatenated to obtain the big key.

We start with the following result to show that for any constant over B^n, there is a polynomial MBA expression computing that constant.

Proposition 2. *For any constant K in $\mathsf{BA}[n]$ and any positive integer m there is a multiterm polynomial* MBA *function f with $f(x_1, \ldots, x_m) = K$ for arbitrary $x_1, \ldots, x_m$ in $\mathsf{BA}[n]$.*

Proof. We use quadratic polynomial and linear MBA identities to construct f. By Theorem 3, with $a \neq 0$, we have an invertible polynomial $p(x) = ax^2 + bx + c$ and its inverse $q(x) = \alpha x^2 + \beta x + \gamma$. Theorem 1 tells us that any singular matrix of size $2^m \times t$, with s, t positive integers, produces an m-variable linear MBA identity. Let $\sum_{i=1}^{r} a_i\, e_i = 0$ be an identity with multiple non-zero terms. Then $K = q(p(K)) = q(\sum_{i=1}^{r} a_i\, e_i + p(K))$. Expanding the expression after rearranging terms in $\sum_{i=1}^{r} a_i\, e_i + p(K)$ yields an m-variable polynomial MBA expression. (This construction is one of many due to $\mathsf{BA}[n]$'s rich mathematical structure.) □

The variable inputs obfuscate the code, and its MBA expression frustrates simplification. The variables are typically shared with the remainder of the application. Appendix A gives a practical example.

4.2 Algorithm and Data Hiding Example: Software Watermarking

As noted in §2.3, systems such as AACS can greatly benefit from MBA-based algorithm and data hiding. As another example, we now show that they can provide a secret watermark to prove identity of software[6], where a watermark is an n-bit constant W, its extraction key is a k-bit constant K, where k is large, and we must embed W in program $\mathbf{P}$ so that we can reliably extract it under K, but an attacker (ignorant of K) cannot obliterate it. (To make n and k large, we can compose W from $W_1, \ldots, W_w$ and K from $K_1, \ldots, K_w$; i.e., we can use multiple sub-watermarks and multiple corresponding sub-keys. The methods should be obvious: we need not discuss this further.)

Watermark Injection. We can represent W and $2^n - W$ as multiterm polynomial MBA expressions by Proposition 2:

$$W = f(x_1, \ldots, x_p) = \sum_{s \in S} a_s \prod_{j \in J_s} e_{s,j}(x_1, \ldots, x_p),$$

$$2^n - W = g(y_1, \ldots, y_q) = \sum_{i \in I} a_i \prod_{j \in J_i} e_{i,j}(y_1, y_2, \ldots, y_q),$$

for any integer variables $x_1, \ldots, x_p,\ y_1, \ldots, y_q \in \mathsf{B}^n$.

Suppose **P** has an MBA expression

$$h(z_1, \ldots z_r) = \sum_{t \in T} a_t \prod_{j \in J_t} e_{t,j}(z_1, \ldots z_r),$$

as an intermediate computation. For example, h could be any single BA operation, which can be represented by multiterm polynomial MBA expressions due to Theorem 3.

We can embed watermark W into h based on $h = W + h + (2^n - W)$. WLOG, assume index sets I, S and J are disjoint. Since $x_1, \ldots, x_p,\ y_1, \ldots, y_q$ are random variables, we can choose some of them from the set $\{z_1, \ldots, z_r\}$. Therefore, the intersection of variable sets $\{y_1, \ldots, y_q\} \cap \{x_1, \ldots, x_p\} \cap \{z_1, \ldots, z_r\}$ maybe non-empty. Represent all with the new set

$$\{u_1, \ldots, u_w\} = \{y_1, \ldots, y_q\} \cup \{x_1, \ldots, x_p\} \cup \{z_1, \ldots, z_r\}.$$

Let σ be any permutation of index set $I \cup S \cup J$. We have

$$h = \sum_{\sigma(t) \in I \cup S \cup J} a_{\sigma(t)} \prod_{j \in J_{\sigma(t)}} e_{\sigma(t),j}(u_1, u_2, \cdots, u_w).$$

Watermark W is computed by adding all terms with indices in $\sigma(S)$ and permutation σ is the watermark key K. To further obfuscate the watermark, apply permutation polynomial p to the watermark expression to mix its terms with terms in h. Then the table-concatenation of permutation σ and the the inverse permutation polynomial p^{-1}'s coefficients is watermark key K.

Following this general watermarking method, we can embed constant watermarks into any BA operations and MBA expressions. The embedding is stealthy because recovering a watermark without the key is an instance of the NP-complete Subset Sum problem, and can be made hard in practice by applying the protections of §3.4.

Watermark Extraction. For any program **P** there are likely to be inputs which are extremely improbable in normal use. Let **P** be a program containing an injected watermark W under key K as described above. At this point, extracting W from the executable code is awkward, and the techniques of this very paper could be used to obscure it and make it effectively impossible to extract.

Instead, the program extracts its own watermark by reüsing the above polynomial computations. (Generating code to do this which we are *building* the code is easy.) Let an extremely improbable lineup of inputs for **P** be **K**. Find an encoding or encodings E according to §3.4 such that $E(\mathbf{K}) = K$ (an easy problem). Then, replace **P** with $\mathbf{P}'$ which, if its input-lineup $\mathbf{I} = \mathbf{K}$, computes $K = E(\mathbf{K})$ and uses K to extract and output W, but in all other circumstances, computes $\mathbf{P}(\mathbf{I})$ as in the original program (see Fig. 3).

Now, encode $\mathbf{P}'$ according to §3.4, using interlocking to render the watermarking aspects of the code highly fragile under tampering, and making the normal behavior of $\mathbf{P}'$ highly dependent on the watermarking code, producing

final program **Q**, the final watermark-protected version of **P**, with a watermark W protected under the new encoded key **K**.

Extensions to interactive or transaction programs and the like are straightforward and left as an exercise.

Watermark Robustness. The watermark in final watermarked program **Q** above is protected as follows. The attacker does not know **K** and so cannot directly attack the response to **K**. Attempting to otherwise change the I/O behavior of the program produces chaotic results: useless to the attacker since it simply destroys the program instead of obliterating the watermark. Further complicating the program by the methods in this paper does not obliterate the watermark, since so protecting programs by transforms and identities preserves functionality and the watermark extraction facility is a (very well hidden) part of that functionality. Of course, the attacker could add another such watermark, but that would not obliterate the original one.

5 Security of MBA Transforms

By experiment, analytical math tools (**Mathematica**$^{\text{TM}}$, **Maple**$^{\text{TM}}$) can't simplify most MBA expressions. Moreover, the following problems are NP-hard:

1. $\mathsf{BA}[n]$-SAT: for a given polynomial MBA function $f(x_1, \ldots, x_m)$ over $\mathsf{BA}[n]$, find values $a_1, \ldots, a_m$ such that $f(a_1, \cdots, a_m) \neq 0$;
2. $\mathsf{BA}[n]$-RECOG: for polynomial MBA functions $f(x_1, \ldots, x_m)$, $g(x_1, \ldots, x_m)$ over $\mathsf{BA}[n]$, find values $a_1, \ldots, a_m$ such that $f(a_1, \ldots a_m) \neq g(a_1, \ldots, a_m)$.

Proposition 3. $\mathsf{BA}[n]$-SAT *and* $\mathsf{BA}[n]$-RECOG *are* NP*-complete.*

Proof. The result follows trivially from NP-completeness of Boolean SAT. □

(Efficient Boolean SAT-solvers can't directly solve these. Boolean SAT is a small subset of $\mathsf{BA}[n]$-SAT: the above problems require solvers for vectors of mutually constrained Boolean SAT-problems.)

Average-case complexity theory is too preliminary for direct security proofs: in the *Average Case Complexity Forum*[2], the papers list was last updated in March of 2000; of roughly 1200 papers added to the *Electronic Colloquium on Computational Complexity*[8] since 1994, about one percent seem to be on average case complexity. We must argue our case by other means.

The best available theory on MBAs is provided in this very paper. We provide easy ways to complicate, but *not* to simplify, programs.

Consider a program whose computations are *interlocked* and *encoded* according to §3.4. Then consider any subexpression repeatedly transformed via the constructions of Theorems 1 and 2. Plainly the program interlocking and encoding and have vastly many choices, and by the nature of the constructions the identity-substitutions can each have vastly many choices as well.

The simplification problem for such programs is the problem of reversing the above process; the vast numbers of choices at every step make its search space

vast. Indeed, as expressions grow during this process, the increase in choices with increasing numbers of steps is *hyperexponential*. We don't expect efficient simplifiers for such massively complex constructions any time soon. **Maple**^{TM}, **Mathematica**^{TM}, or similar foreseeable tools, need large databases of known identities and laws. As we have argued above, for MBAs, we should change 'large' to 'extremely large', and probably to 'infeasibly large'; methods depending on simple laws such as Karnaugh-map Boolean simplification are inadequate. New characterizations of NP problems, such as PCP, might eventually offer attack methods — but probably not soon.

Without an efficient simplifier, the limitless variety of polynomial MBA representations for any given BA operation, and the code composed of such operations, becomes an insurmountable BA$[n]$-RECOG function recognition burden for the attacker: the attacker must recognize the code functionality, but the search space for the attacker is vast and grows over time as increasing numbers of transforms are generated.

We can apply other protections before and/or after those in this paper: see [14] for an excellent survey. Such techniques are important to provide defense in depth and to address other attacks, for example, tampering or spoofing of system calls. This paper's methods proposed ensure viability of such additional protections by obviating any need for specialized hardware.

6 Conclusion

We have introduced powerful new methods for protecting constants, data, and code in software by converting computations into mixed Boolean-arithmetic computations, whether linear or polynomial, over BA-algebras, which provide a rich algebraic system based on the functionality of real-world computer instructions. Such protections are not penetrable using existing analytical tools such as **Maple**^{TM} or **Mathematica**^{TM}, since they combine multiple algebraic systems into one exceedingly complex system.

By basing our protections on ordinary computer instructions, we avoid the vulnerabilities of methods of protection which require code to appear in unexecutable forms or in forms which require an interpreter or a substantial supporting library.

These MBA transform methods provide open-ended generators for identities and transformations which provide an effectively *unlimited* supply of encodings for data and code, making the search-space for attackers very large, and increasingly large over time, as we accumulate more and more vast sets of identities and transforms.

We have described methods for hiding constants in code and for protecting code by keys, so that tampering with the code, with high probability, causes the code to malfunction, and given practical examples to demonstrate our methods.

Acknowledgements. The authors thank their colleagues Phil Eisen, Clifford Liem, and the anonymous referees, for valuable comments.

References

1. Arora, S., Safra, S.: Probabilistic checking of proofs: A new characterization of NP. Journal of the ACM 45(1), 70–122 (1998)
2. Average Case Complexity Forum, http://www.uncg.edu/mat/avg.html
3. Chow, S., Johnson, H., Gu, Y.X.: Tamper resistant software encoding, US Patent No. 6594761 (2003)
4. Chow, S., Gu, Y.X., Johnson, H., Zakharov, V.A.: An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In: Davida, G.I., Frankel, Y. (eds.) ISC 2001. LNCS, vol. 2200, pp. 144–155. Springer, Heidelberg (2001)
5. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.C.: White-Box Cryptography and an AES Implementation. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, Springer, Heidelberg (2003)
6. Collberg, C.S., Thomborson, C.: Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. IEEE Trans. Software Eng. 28(6) (June 2002)
7. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.H.: Oblivious Hashing: A Stealthy Software Integrity Verification Primitive. In: Petitcolas, F.A.P. (ed.) IH 2002. LNCS, vol. 2578, pp. 400–414. Springer, Heidelberg (2003)
8. Electronic Colloquium on Computational Complexity, http://eccc.hpi-web.de/eccc/ ISSN 1433-8092
9. Garey, M.R., Johnson, D.S.: Computers and Intractability. W.H. Freeman and Company, New York (1979)
10. Kandanchatha, A.N., Zhou, Y.: System and method for obscuring bit-wise and two's complement integer computations in software, Canadian patent application 2456644, 2004; US Patent Application 20050166191 (2005)
11. Klimov, A., Shamir, A.: Cryptographic Applications of T-functions. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, pp. 248–261. Springer, Heidelberg (2004)
12. Klimov, A.: Applications of T-functions in Cryptography, PhD Thesis, Weizmann Institute of Science (2004)
13. Mullen, G., Stevens, H.: Polynomial functions (mod m). Acta Mathematica Hungarica 44(3-4), 287–292 (1984)
14. van Oorschot, P.C.: Revisiting Software Protection. In: Boyd, C., Mao, W. (eds.) ISC 2003. LNCS, vol. 2851, pp. 1–13. Springer, Heidelberg (2003)
15. Rivest, R.L.: Permutation Polynomials Modulo 2^w. Finite Fields and their Applications 7, 287–292 (2001)
16. Sander, T., Tschudin, C.F.: Towards Mobile Cryptography. In: Proceedings of the 1998 IEEE Symposium on Security and Privacy, pp. 215–224 (1998)
17. Sander, T., Tschudin, C.F.: Protecting Mobile Agents Against Malicious Hosts. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 44–60. Springer, Heidelberg (1998)
18. Ogiso, T., Sakabe, Y., Soshi, M., Miyaji, A.: Software Tamper Resistance Based on the Difficulty of Interprocedural Analysis. In: Proceedings of WISA 2002 (2002)
19. Vuillemin, J.: Digital algebra and Circuits. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, Springer, Heidelberg (2004)
20. Warren Jr., H.S.: Hacker's Delight. Addison-Wesley, Boston (2002), www.hackersdelight.org
21. Zhou, Y., Main, A.: Diversity via Code Transformations: A Solution for NGNA Renewable Security, The NCTA Technical Papers^{TM} 2006, The National Cable and Telecommunications Association Show, Atlanta, pp. 173–182 (2006)
22. http://www.aacsla.com

A Example of Key Hiding in an MBA Polynomial

Let the key be $K = 0x87654321$ (hex). We use three input variables $x, x_1, x_2 \in \mathsf{B}^{32}$, two linear MBA identities

$$2y = -2(x \vee (-y-1)) - ((-2x-1) \vee (-2y-1)) - 3;$$
$$x + y = (x \oplus y) - ((-2x-1) \vee (-2y-1)) - 1;$$

and one polynomial transform

$$f(x) = 727318528x^2 + 3506639707x + 6132886 \in P_2(\mathsf{Z}/(2^{32}))$$

to generate the following key code:

$$\begin{aligned}
a =\ & x(x_1 \vee 3749240069); b = x((-2x_1 - 1) \vee 3203512843);\\
d =\ & ((235810187x + 281909696 - x_2) \oplus (2424056794 + x_2));\\
e =\ & ((3823346922x + 3731147903 + 2x_2) \vee (3741821003\\
+\ & 4294967294x_2));\\
key =\ & 4159134852e + 272908530a + 409362795x + 136454265b\\
+\ & 2284837645 + 415760384a^2 + 415760384ab + 1247281152ax\\
+\ & 2816475136ad + 1478492160ae + 3325165568b^2 + 2771124224bx\\
+\ & 1408237568bd + 2886729728be + 4156686336x^2 + 4224712704xd\\
+\ & 70254592xe + 1428160512d^2 + 1438646272de + 1428160512e^2\\
+\ & 135832444d,
\end{aligned}$$

where $a, b, d, e \in \mathsf{B}^{32}$ are intermediate variables. The output value of key is always the constant K regardless of values in x, x_1, and x_2.